Pattern matching

From Wikipedia, the free encyclopedia.

Pattern matching is the act of checking for the presence of the constituents of a given pattern. In contrast to pattern recognition, the pattern is rigidly specified. Such a pattern concerns conventionally either sequences or tree structures. Pattern matching is used to check that things have the desired structure, to find relevant structure, to retrieve the aligning parts, and to substitute the matching part with something else.

Sequence (or specifically text string) patterns are often described using regular expressions and matched using respective algorithms. Sequences can also be seen as trees branching for each element into the respective element and the rest of the sequence, or as trees that immediately branch into all elements.

Tree patterns can be used in programming languages as a general tool to process data based on its structure. Some functional programming languages such as Haskell, ML and the symbolic mathematics language Mathematica have a special syntax for expressing tree patterns and a language construct for conditional execution and value retrieval based on it. Because of simplicity and efficiency reasons these tree patterns lack some features that are available in regular expressions. Depending on the languages, pattern matching can be used for function arguments, in case expressions, whenever new variables are bound, or in very limited situations such as only for sequences in assignment in Python. Often it is possible to give alternative patterns that are tried one by one. Pattern matching can benefit from guards.

Term rewriting languages rely on pattern matching for the fundamental way a program evaluates into a result. Pattern matching benefits most when the underlying datastructures are as simple and flexible as possible. This is especially the case in languages with a strong symbolic flavor. In symbolic programming languages, patterns are the same kind of datatype as everything else, and can therefore be fed in as arguments to functions.

Contents

Primitive patterns

The simplest pattern in pattern matching is an explicit value or a variable. For an example, consider a simple function definition in Haskell syntax (function parameters are not in parentheses but are separated by spaces, = is not assignment but definition):

```
f 0 = 1
```

EXHIBIT A

Here, 0 is a single value pattern. Now, whenever f is given 0 as argument the pattern matches and the function returns 1. With any other argument, the matching and thus the function fail. As the syntax supports alternative patterns in function definitions, we can continue the definition extending it to take more generic arguments:

```
f n = n * f (n-1)
```

Here, the first `n` is a single variable pattern, which will match absolutely any argument and bind it to name n to be used in the rest of the definition. In Haskell (unlike at least Hope), patterns are tried in order so the first definition still applies in the very specific case of the input being 0, while for any other argument the function returns `n * f (n-1)` with n being the argument.

Wildcard pattern (often written as _) is also simple: like a variable name, it matches any value, but does not bind the value to any name.

Tree patterns

More complex patterns can be built from the primitive ones of the previous section, usually in the same way as values are built by combining other values. The difference then is that with variable and wildcard parts, a pattern doesn't not build into single value, but matches a group of values that are the combination of the concrete elements and the elements that are allowed to vary within the structure of the pattern.

A tree pattern describes a part of a tree by starting with a node and specifying some branches and nodes and leaving some unspecified with a variable or wildcard pattern. It may help to think of the abstract syntax tree of a programming language and algebraic data types.

In Haskell, the following line defines an algebraic data type `Color` that has a single data constructor `ColorConstructor` that wraps an integer and a string.

```
data Color = ColorConstructor Integer String
```

The constructor is a node in a tree and the integer and string are leaves in branches.

When we want to write functions to make `Color` an abstract data type, we wish to write functions to interface with the data type, and thus we want to extract some data from the data type, for example, just the string or just the integer part of `Color`.

If we pass a variable that is of type Color, how can we get the data out of this variable? For example, for a function to get the integer part of `Color`, we can use a simple tree pattern and write:

```
integerPart (ColorConstructor theInteger _) = theInteger
```

Or conversely:

```
stringPart (ColorConstructor _ theString) = theString
```

Filtering data with patterns

Pattern matching can be used to filter data of a certain structure. For instance, in Haskell a list comprehension could be used for this kind of filtering:

```
[A x | A x <- [A 1, B 1, A 2, B 2]]
```

evaluates to

```
[A 1, A 2]
```

Pattern matching in Mathematica

In Mathematica, the only structure that exists is the tree, which is populated by symbols. In the Haskell syntax used thus far, this could be defined as

```
data SymbolTree = Symbol String [Symbol]
```

An example tree could then look like

```
Symbol "a" [Symbol "b" [], Symbol "c" []]
```

In the traditional, more suitable syntax, the symbols are written as they are and the levels of the tree are represented using [], so that for instance a[b,c] is a tree with a as the parent, and b and c as the children.

A pattern in Mathematica involves putting "_" at positions in that tree. For instance, the pattern

```
A[_]
```

Will match elements such as A[1], A[2], or more generally A[x] where x is absolutely any entity. In this case, A is the concrete element, while _ denotes the piece of tree that can be varied. A symbol prepended to _ binds the match to that variable name while a symbol appended to _ restricts the matches to nodes of that symbol.

Mathematica function Cases filters elements of the first argument that match the pattern in the second argument:

```
Cases[{a[1], b[1], a[2], b[2]}, a[_] ]
```

evalutes to

```
{a[1], a[2]}
```

Pattern matching applies to the *structure* of expressions. In the example below,

```
Cases[{a[b], a[b,c], a[b[c], d], a[b[c], d[e]], a[b[c], d, e]}, a[b[_],_]]
```

returns

```
{a[b[c],d], a[b[c],d[e]]}
```

because only these elements will match the pattern `a[b[_],_]` above.

In Mathematica, it is also possible to extract structures as they are created in the course of computation, without regard to how or where they appear. The function Trace can be used to monitor a computation, and return the elements that arise which match a pattern. For example, we can define the Fibonnaci sequence as

```
fib[0|1]:=1
fib[n_]:= fib[n-1] + fib[n-2]
```

Then, we can ask the question: Given fib[3], what is the sequence of recusive Fibonacci calls?

```
Trace[fib[3], fib[_]]
```

returns a structure that represents the occurrences of the pattern fib[_] in the computational structure:

```
{fib[3],{fib[2],{fib[1]},{fib[0]}},{fib[1]}}
```

Declarative programming

In symbolic programming languages, it is easy to have patterns as arguments to functions or as elements of datastructures. A powerful consequence of this is the ability to use patterns to declaratively make statements about pieces of data and to flexibly instruct functions how to operate.

For instance, the Mathematica function Compile can be used to make highly efficient versions of the code. In the following example the details do not particularly matter; what matters is that the subexpression {{com[_], _Integer}} instructs Compile that expressions of the form com[_] can be assumed to be integers for the purposes of compilation:

```
com[i_] := Binomial[2i, i]
Compile[{x, {i, _Integer}}, x^com[i], {{com[_], _Integer}}]
```

Pattern matching and strings

By far the most common form of pattern matching involves strings of characters. In many programming

languages, a particular syntax of strings is used to represent regular expressions, which are patterns describing string characters.

However, it is possible to perform some string pattern matching within the same framework that has been discussed throughout this article.

Tree patterns for strings

In Mathematica, strings are represented as trees of root StringExpression and all the characters in order as children of the root. Thus, to match "any amount of trailing characters", a new wildcard ___ is needed in contrast to _ that would match only a single character.

In Haskell and functional programming languages in general, strings are represented as functional lists of characters. A functional list is defined as an empty list, or an element constructed on an existing list. In Haskell syntax:

```
[]      -- an empty list
x:xs    -- an element x constructed on a list xs
```

The structure for a list with some elements is thus `element:list`. When pattern matching, we assert that a certain piece of data is equal to a certain pattern. For example, in the function:

```
head (element:list) = element
```

we assert that the first element of `head`'s argument is called element, and the function returns this. We know that this is the first element because of the way lists are defined, a single element constructed onto a list. This single element must be the first. The empty list would not match the pattern at all, as an empty list does not have a head (the first element that is constructed).

In the example, we have no use for `list`, so we can disregard it, and thus write the function:

```
head (element:_) = element
```

The equivalent Mathematica transformation is expressed as

```
head[element_, ___]:=element
```

Example string patterns

In Mathematica, for instance,

```
StringExpression["a", _]
```

will match a string that has two characters and begins with "a".

The same pattern in Haskell:

```
['a', _]
```

Symbolic entities can be introduced to represent many different classes of relevant features of a string. For instance,

```
StringExpression[LetterCharacter, DigitCharacter]
```

will match a string that consists of a letter first, and then a number.

In Haskell, guards could be used to achieve the same matches:

```
[letter, digit] | isAlpha letter && isDigit digit
```

The main advantage of symbolic string manipulation is that it can be completely integrated with the rest of the programming language, rather than being a separate, special purpose subunit. The entire power of the language can be leveraged to built up the patterns themselves or analyze and transform the programs that contain them.

See also

- Pattern recognition for fuzzy patterns
- Category:Pattern matching for articles about string pattern matching
- AIML for an AI language based on matching patterns in speech

External links

- A Gentle Introduction to Haskell: Patterns (http://www.haskell.org/tutorial/patterns.html)
- Views: An Extension to Haskell Pattern Matching (http://www.haskell.org/development/views.html)
- Nikolaas N. Oosterhof, Philip K. F. Hölzenspies, and Jan Kuper. Application patterns (http://wwwhome.cs.utwente.nl/~tina/apm/applPatts.pdf). A presentation at Trends in Functional Programming, 2005

References

- The Mathematica Book, chapter Section 2.3: Patterns (http://documents.wolfram.com/mathematica/book/section-2.3)
- The Haskell 98 Report, chapter 3.17 Pattern Matching (http://haskell.org/onlinereport/exps.html#pattern-matching).
- Pattern matching (http://ftp.sunet.se/foldoc/foldoc.cgi?pattern+matching) in The Free On-line Dictionary of Computing, Editor Denis Howe.

- Python Reference Manual, chapter 6.3 Assignment statements (http://python.org/doc/2.4.1/ref/assignment.html).

Retrieved from "http://en.wikipedia.org/wiki/Pattern_matching"

Category: Programming constructs